International Computer Science Competition

Solutions for the Qualification Round
by Maia Curti Pérez
August-September 2025

Problem A: Neural Network Components

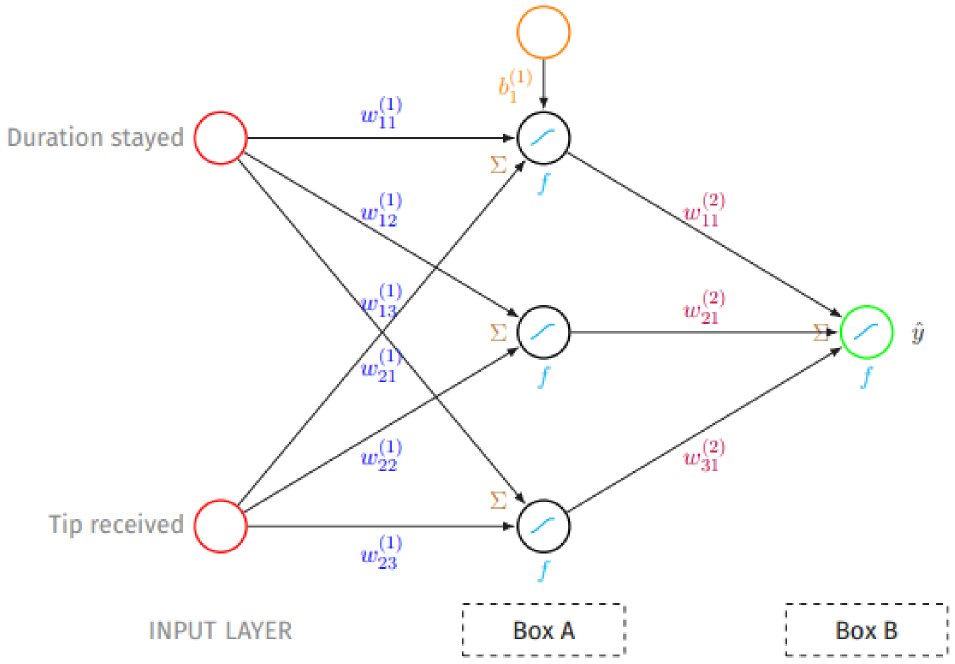


Figure 1: Illustration of a neural network (MLP) that predicts whether a customer was satisfied with their visit (provided by ICSC).

$\omega_{21}^{(1)}$ represents the weight (how strong the connection between the neurons is). The input from the second neuron (Tip recieved) is miltuplied by $\omega_{21}^{(1)}$ and then added to the calculated weight from $\omega_{11}^{(1)}$ in the first neuron of the hidden layer.

$\sum$ is a sumation. The calculated weight for each input is added. In the first neuron of the hidden layer, the weights corresponding to $\omega_{11}^{(1)}$ and $\omega_{21}^{(1)}$ are added.

f represents the activation function. One example of activation functions is ReLU: it changes any negative number to 0 and it does not change the positive numbers.

The red circles represent input neurons.

The orange circle represents the bias corresponding to the first neuron of the hidden layer.

The green circle represents the output neuron.

Box A represents a hidden layer (the only one in this example).

Box B represents the output layer.

$\hat{y}$ represents the calculated probability that the customer was satisfied with their visit.

Problem B: Cake Calculator

Below is the implementation of the algorithm (in Python) used to calculate the amount of cakes and the amount of flour and sugar remaining when given the units of flour and sugar available. The skeleton code provided by ICSC is not showed here. The complete file (skeleton code with the solution) has also been submitted.

```python
# Set the units of flour and sugar needed for each cake
flourNeeded = 100
sugarNeeded = 50

# Set the counter that will keep track of the amount of cakes made
cakeCounter = 0

# Calculate the amount of cakes that can be made with the ingredients available
while flour >= flourNeeded and sugar >= sugarNeeded:
  flour-=flourNeeded
  sugar-=sugarNeeded
  cakeCounter+=1

# Set the amount of flour and sugar left
flourLeft = flour
sugarLeft = sugar

# Return the amount of cakes that can be made, the leftover units of flour
# and the leftover units of sugar.
return [cakeCounter, flourLeft, sugarLeft]
```

The algorithm first sets the units of flour and sugar that are needed to make a cake. It also sets a counter for the cakes made (starts at 0). Then, as long as there is enough flour and sugar for a cake, it subtracts the amount of flour and sugar needed for the cake from the available units and adds one to the cake counter. Once there is not enough flour or not enough sugar it stops the loop and returns the cakes that can be made, the leftover units of flour and sugar.

The last lines (14-20) could be replaced by the following. The first version helps with readability, while the second one avoids creating extra variables.

```
    # Return the amount of cakes that can be made, the leftover units of flour
    # and the leftover units of sugar.
    return [cakeCounter, flour, sugar]
```

Problem C: The School Messaging App

Question 1

Why would using different length codes for characters with different probabilities help you transmit more messages within your data limit? Give an example.

Using shorter codes for characters with more probability would, on average, reduce the number of bits used: the characters that appear more often require fewer bits than the rarer ones. Therefore, it would help to transmit longer messages within the limit.

For example, using 4 bits per character, the average code length for the 12-character set given would be 4. Using 3 bits for characters with a probability higher than 0.10, 4 bits for the other characters, the average code length would be 3.53.

Question 2

Calculate the entropy for the 12-character set above. What does this value represent in terms of optimal encoding?

The entropy for the 12-character set above is $H = 3.324$ (rounded to the third decimal), indicating that, on average, the minimum number of bits needed to encode messages is 4.

Question 3

Calculate the average code length of your Fano code and compare it to the theoretical entropy limit. How efficient is your Fano code?

The average code length of the Fano code is 3.48, it is close to the entropy value (only a couple of decimals higher). This means it conveys the information in a very efficient way: the number of bits it uses is, on average, close to the theoretical minimum (H).

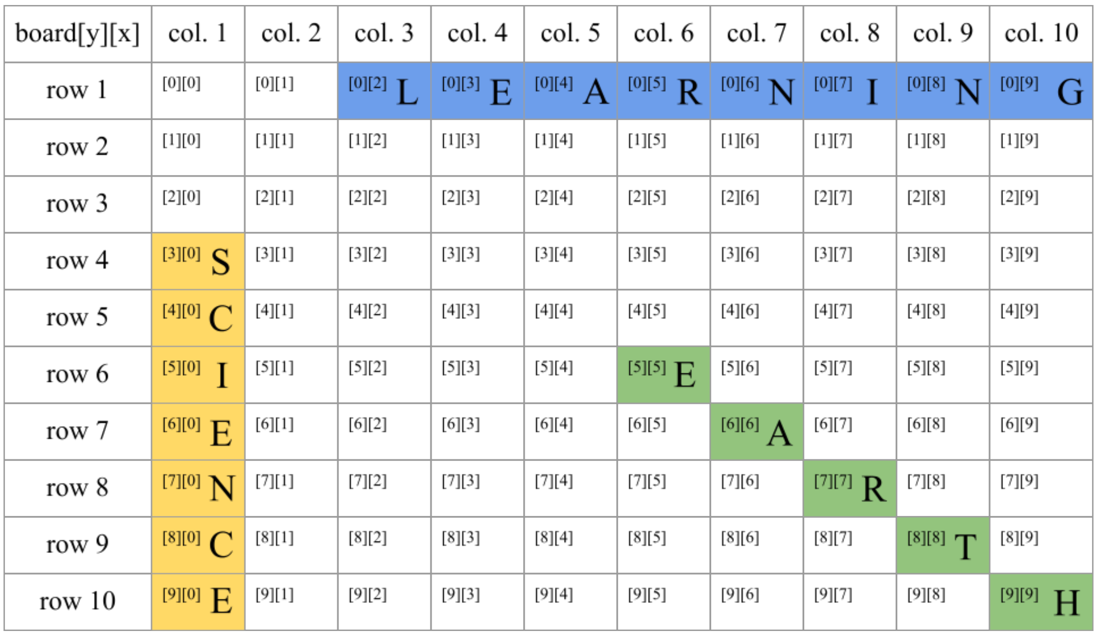
Problem D: Word Search Puzzle

Below is the implementation of the algorithm to generate a 10x10 word search puzzle when given a list of words. The skeleton code provided by ICSC is not showed here. The complete file (skeleton code with the solution) has also been submitted.

```python
# placing the longer words first helps with speed and
# lowers the chances of running into problems at the end
words.sort(key=len, reverse=True)

# create the board
board = []
for i in range(10):
    board.append([])
    for _ in range(10):
        board[i].append("")

# place each word in the board
for word in words:
    word = word.upper() # change the letters to uppercase
    orientation = random.randint(0, 2) # 0=horizontal, 1=vertical, 2=diagonal

    if orientation == 0: # horizontal
        while True:
            check = 0
            x = random.randint(0, 10-len(word)) # making sure it fits
            y = random.randint(0, 9)
            for i in range(len(word)):
                if board[y][x+i] == "" or board[y][x+i] == word[i]:
                    check += 1
            if check == len(word):
                # there is no incompatibility between the board and the word
                for i in range(len(word)):
                    board[y][x+i] = word[i]
                break

    elif orientation == 1: # vertical
        while True:
            check = 0
            x = random.randint(0, 9)
            y = random.randint(0, 10-len(word)) # making sure it fits
            for i in range(len(word)):
                if board[y+i][x] == "" or board[y+i][x] == word[i]:
                    check += 1
            if check == len(word):
                # there is no incompatibility between the board and the word
                for i in range(len(word)):
                    board[y+i][x] = word[i]
                break
    else: # diagonal
        while True:
            check = 0
            x = random.randint(0, 10-len(word)) # making sure it fits
            y = random.randint(0, 10-len(word)) # making sure it fits
            for i in range(len(word)):
                if board[y+i][x+i] == "" or board[y+i][x+i] == word[i]:
                    check += 1
            if check == len(word):
                # there is no incompatibility between the board and the word
                for i in range(len(word)):
                    board[y+i][x+i] = word[i]
                break

# fill the rest of the board with random uppercase letters
for i in range(len(board)):
    for j in range(len(board[i])):
        if board[i][j] == "":
            board[i][j] = chr(random.randint(65, 90))

return board
```

The algorithm starts by sorting the list of words from longest to shortest. I decided to do this because, while testing, if the last words were long, many times the algorithm could not find a place to put them.

Then it creates the board: a list of 10 lists with 10 elements. To represent that the places are "empty", the elements are empty strings. A diagram of an example board is shown in fig. 2



board[y][x]	col. 1	col. 2	col. 3	col. 4	col. 5	col. 6	col. 7	col. 8	col. 9	col. 10
row 1	[0][0]	[0][1]	[0][2] L	[0][3] E	[0][4] A	[0][5] R	[0][6] N	[0][7] I	[0][8] N	[0][9] G
row 2	[1][0]	[1][1]	[1][2]	[1][3]	[1][4]	[1][5]	[1][6]	[1][7]	[1][8]	[1][9]
row 3	[2][0]	[2][1]	[2][2]	[2][3]	[2][4]	[2][5]	[2][6]	[2][7]	[2][8]	[2][9]
row 4	[3][0] S	[3][1]	[3][2]	[3][3]	[3][4]	[3][5]	[3][6]	[3][7]	[3][8]	[3][9]
row 5	[4][0] C	[4][1]	[4][2]	[4][3]	[4][4]	[4][5]	[4][6]	[4][7]	[4][8]	[4][9]
row 6	[5][0] I	[5][1]	[5][2]	[5][3]	[5][4]	[5][5] E	[5][6]	[5][7]	[5][8]	[5][9]
row 7	[6][0] E	[6][1]	[6][2]	[6][3]	[6][4]	[6][5]	[6][6] A	[6][7]	[6][8]	[6][9]
row 8	[7][0] N	[7][1]	[7][2]	[7][3]	[7][4]	[7][5]	[7][6]	[7][7] R	[7][8]	[7][9]
row 9	[8][0] C	[8][1]	[8][2]	[8][3]	[8][4]	[8][5]	[8][6]	[8][7]	[8][8] T	[8][9]
row 10	[9][0] E	[9][1]	[9][2]	[9][3]	[9][4]	[9][5]	[9][6]	[9][7]	[9][8]	[9][9] H

Figure 2: Diagram of a board to explain the algorithm to solve problem D.

On line 13 it starts putting each word in the board: it changes the letters to capitals and randomizes the direction (horizontal, vertical, or diagonal) for the word. Then, the algorithm randomizes the starting position (x,y) for the word, making sure that it fits on the board with the chosen direction (see fig. 2):

- The word "learning", a horizontal word, 8 letters long: the starting position cannot be after the third column; in other words, x<=2=10-8.
- The word "science", a vertical word, 7 letters long: the starting position cannot be lower than the fourth row; in other words, y<=3=10-7.
- The word "earth", a diagonal word, 5 letters long: the starting position cannot be after the sixth column or lower than the sixth column; in other words, x,y<=5=10-5.

Having a position candidate (x,y), the algorithm checks that the elements that the word will occupy are empty or equal to the letter the word needs in that space. If there is a letter that does not match the letter the word needs there, the algorithm randomizes a new position candidate until it finds one compatible with the current board.

Once every word has been placed, the algorithm assigns random letters to every empty place left.

I decided that words would not appear backwards (for example, horizontally with the last letter of the word on the left). The following code could be added right bellow line 15 to allow backward words:

```
    orientation = random.randint(0, 1) # 0 = forward, 1 = backward
    if orientation == True:
        word = word[::-1]
```

These lines of code randomize the orientation of the word; then, if backward is chosen, save the word with the new orientation.

Problem E: Functional Completness of NAND

A logic gate (or a set of gates) is said to be functionally complete if any Boolean function can be constructed using only that gate (or gates); the NAND gate outputs 0 only when $a = b = 1$ (thus, it outputs 1 if at least one input is 0).

I have only seen proper logic gates before in Minecraft redstone, so I will take the visual approach instead of the written one. The figures were made with the website *https://logic.ly/demo/*.

The fig. 3 shows how a NOT gate can be expressed as a single NAND gate.

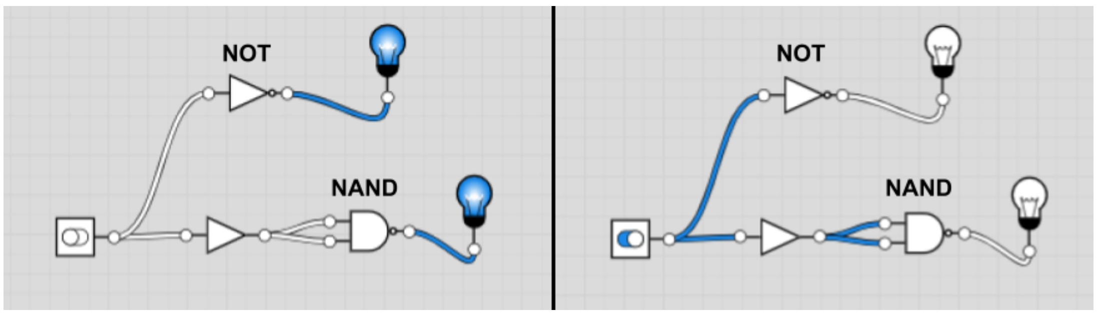


Figure 3: NOT gate expressed with a single NAND gate.

The fig. 4 shows how an OR gate can be expressed with NAND gates.

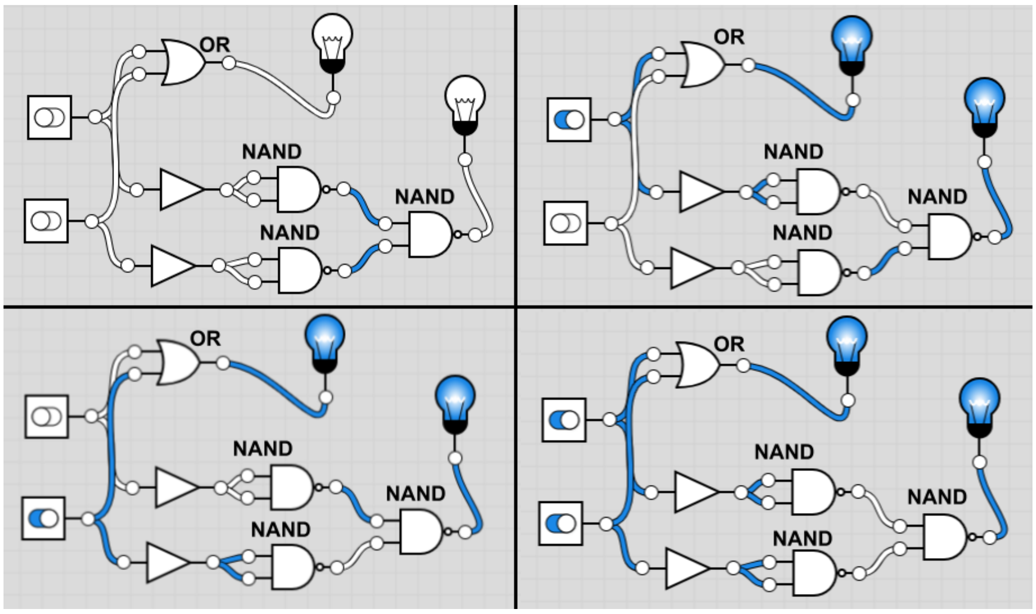


Figure 4: OR gate expressed with NAND gates.

The fig. 5 shows how an AND gate can be expressed with NAND gates.

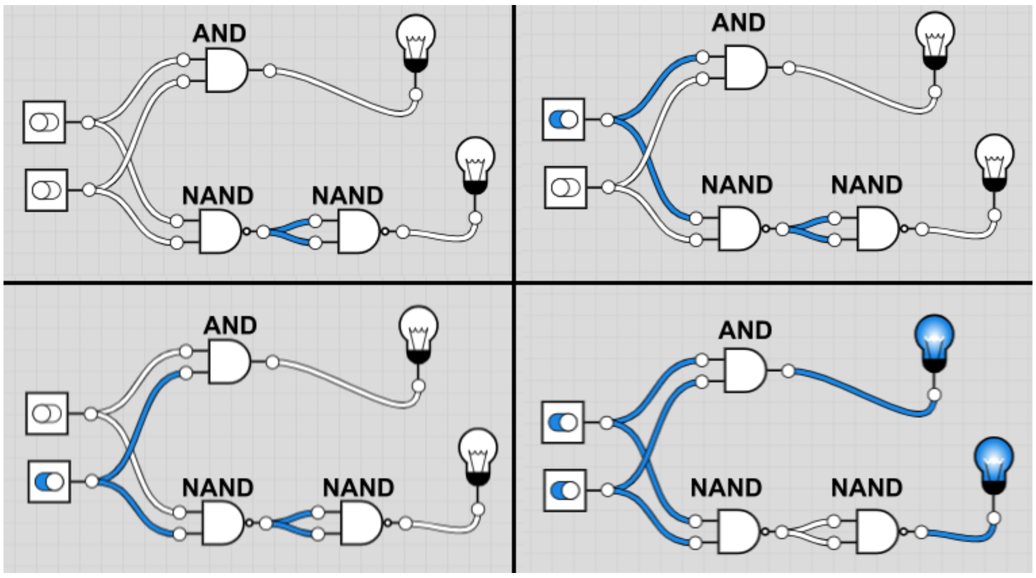


Figure 5: AND gate expressed with NAND gates.

The fig. 6 shows how an XOR gate can be expressed with NAND gates, as an extra.

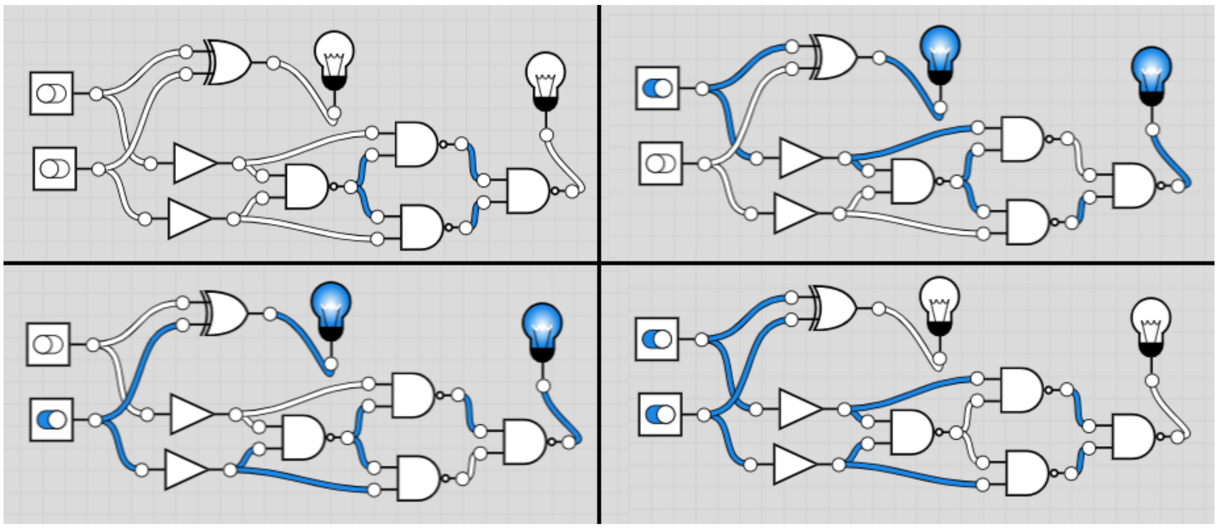


Figure 6: XOR gate expressed with NAND gates.

Being able to express the AND, NOT and OR gates only with NAND gates proves the NAND gate is functionally complete.